Exception or rule?

Rico Huijbers (499062)
2003

Abstract

In this essay, I will discuss the relatively new concept of error handling through “exceptions”, and compare it to traditional techniques. First, I will discuss traditional techniques for error handling, and outline the problems that exist with these techniques. Then I will introduce and discuss exceptions, and see how their usage solves the traditional problems. Finally, I will discuss why traditional techniques might be more useful than exceptions in some rare cases.

1 Introduction

Error handling has traditionally been one of the most important, and paradoxically also one of the most underdeveloped fields of practical programming. That error handling is very important needs little justification: in practice it is more rule than exception that software is subjected to conditions that were not foreseen at development time, and for which regular techniques that establish software correctness do not suffice. The requirement that software behaves gracefully in the face of unknown conditions is also known as “robustness” [3]. To improve robustness, a lot of effort must be put into developing the program and thinking about unforeseen conditions[1], and the best ways to handle them.

In this paper, we will take a look at traditional ways of handling erroneous or exceptional situations. Then we will take a look at the new method of handling them through exceptions, and we will compare the two methods extensively. In [1], Sebesta mentions the qualities *readability, writability* and *reliability* as the most important factors of programming constructs, and programming languages in general. We will compare traditional methods and exceptions with respect to these qualities.

In addition to providing better readability and reliability for our software projects, we will see that exceptions are able to cover for error situations that are not normally detectable by traditional error-handling methods. Not only do they make error handling easier on the part of the programmer, they also extend his error-handling capabilities altogether.

In this paper, when comparing traditional, procedural languages to new, exception-capable ones, I will most often refer to the widely known C and Pascal, and their extended, exception-capable counterparts, C++ and Object Pascal (Delphi).

[1] Unforeseen conditions are also called “exceptional” conditions, hence the term exceptions.

2 Traditional methods of error handling

In traditional (procedural) programming languages, error handling is mostly realized through the inspection of success values, returned by subprograms. This implies two things, namely that error handling involves responding to the success or failure of subprograms, and that subprograms need to be able to return those values. Before error handling, we shall first look at error reporting, which is the way in which subprograms can signify to their caller whether their operation was successful or whether some sort of malfunction occurred.

2.1 Traditional error reporting

Subprograms can usually return values in two ways: A value can be returned in a variable of the actual parameter list, or it can be returned as the result of a function call. Since the use of a variable in the parameter list requires a variable of the appropriate type to be defined, and since it takes two statements to fill and then inspect such a variable, in the interest of writability, most routines are written to be functions and return their success value as the function result. While this does indeed provide a convenient way to return success values, it leads to a pollution of the classical notion that functions and procedures are not the same. Indeed, procedures are subprograms that perform some task, and functions are subprograms that calculate some value without causing side effects. With this method of error reporting, however, all procedures are also encoded as functions and the distinction between them is lost [2].

A subprogram might fail for a lot of reasons. Perhaps the subprogram call was malformed, or contained illegal data. Perhaps the subprogram did not have enough permissions to perform the action it intended to, or some I/O action it performed triggered an error. Since a calling program usually needs to know the kind of error that occurred (if not for correcting the situation itself, then for showing a helpful message to the user of the program, so that he may correct the problem), the error needs to be encoded in some way. A return value can indicate the kind of error that occurred (for example, in the form of some predefined integer or string constant), or it can simply be a Boolean value that signifies whether the subprogram call was successful or not. The exact nature of the error can then usually be looked up in some kind of central error repository, for example through a function call like *GetLastError* [2].

2.2 Traditional error handling

Calling programs can inspect the success values returned by subprograms, and select a branch of execution depending on whether or not the subprogram call was successful. The most common branch selection used is either to keep carrying out the steps of the subprogram's algorithm, or abort whenever an error occurs and indicate failure to the program's caller. If the calling program is itself a subprogram, aborting usually means discontinuing the subprogram's execution and returning an error value. If the calling program is a main program, aborting entails displaying some kind of error message to the program's caller (the user) and then halting program execution.

In some rare conditions, a calling program can decide to change the calling conditions and try the call again, until the call succeeds or enough tries have been made to warrant the guess that the call will not succeed anymore. Whatever the calling program's choice, it is required that the program act upon the success values received from subprograms, either by aborting or trying again. This ensures that the result of actions is never ignored, and increases program reliability and robustness.

In Pascal, a sample call and error check in a main program might look like this:

[2] Since C does not have "pure" procedures, only functions that optionally return nothing, the distinction is not so large there. However the theoretical grounds for rejecting this method remain.

```
Result := PerformOperation(SomeParameters);
if Result = Success then
     WriteLn('Operation was succesful!')
else
     WriteLn('Operation failed: ', Result);
```

or as it is usually written:

```
if PerformOperation(SomeParameters) = Success then
     WriteLn('Operation was successful!')
else
     WriteLn('Operation failed!');
```

Note that in the second case the success value is not saved, and can therefore not be shown to the user. At this point, the user must guess what went wrong inside the subprogram. And even if the return value is saved and displayed, it is usually implemented as an integer constant. Just showing some integer value to the user is almost as bad as not showing any reason at all, since most (regular) users have no information about the error the integer is meant to represent.

Now, let's see what happens when there are multiple steps to execute, each one depending on the previous one's success. This is a very common case in which various initializations of subsystems have to be performed before the actual desired action can be carried out. Written in C, such a sequence might look like this:

```
if (operation1(param1)) {
     if (operation2(param2)) {
          if (operation3(param3)) {
               printf("All operations successful");
          } else
               printf("Operation 3 failed: %d", GetLastError());
     } else
          printf("Operation 2 failed: %d", GetLastError());
} else
     printf("Operation 1 failed: %d", GetLastError());
```

As can be seen in this example, the readability and especially the modifiability of such a structure suffer severely from the nesting of several of such error handling clauses.

These examples show how error handling might look in a main program. In case an error occurs, execution is not continued and a descriptive error message is shown to the user. In subprograms, it is usually undesirable to directly output error information, and subprogram result must be signified to the caller. Again in C, a subprogram that performs initialization of some subsystem might look like this, assuming the initialization required consists of two steps:

```
int initialize() {
     if (subsystem_init_step1() == NO_ERROR) {
          if (subsystem_init_step2() == NO_ERROR) {
               return NO_ERROR;
          } else
               return ERR_STEP2_FAILED;
     } else
          return ERR_STEP1_FAILED;
}
```

As can be seen, although the possible errors occur in `subsystem_init_step1` and `subsystem_init_step2`, we have to check the result value of each subprogram, abort if it failed and let our own success value depend on the result of each subprogram. In the next section, we will see how we can drastically improve on this subroutine through the use of exceptions.

Finally, since the only way to set a success value is to explicitly set or return the appropriate status value through user code, this method of error handling is simply unable to cope with non-software detectable errors (such as hardware failure, or invalid pointer dereferencing).

2.3 Traditional error handling summary

In summary, the problems with traditional error handling are:

- Distinction between procedures and functions is lost
- Integer error values do not carry meaningful information
- Nesting of several handlers detracts greatly from readability and modifiability
- Error propagation must be explicitly implemented at every level of the subprogram call chain
- Traditional handling is unable to cope with certain classes of errors

3 Exceptions

Enter exceptions. Exceptions are the new standard of error handling, which overcome all of the problems of traditional error handling outlined in the previous section.

3.1 What are exceptions?

Exceptions are signals indicating that some kind of erroneous event has occurred during the execution of a program. Exceptions can originate from different sources. They can be generated by user code, the language run-time, the operating system, or even by the hardware. The fact that error conditions detected and signaled from outside the application can be handled through user code is a very powerful improvement. It allows elegant handling of conditions that were not normally detectable in application software, such as the dereferencing of null- or dangling pointers.

Most contemporary languages supporting exceptions also include support for object-orientation. Therefore, although this is not a requirement, in most languages exceptions are implemented as an object. This has the added benefit that various attributes describing the conditions of the error in more detail can be attached to the exception. This information is then available for inspection, or display to the user, at the moment the error condition is handled.

3.2 Error reporting through exceptions

The most important characteristic of exceptions is that they can be *raised*[3]. Raising an exception indicates that the error condition it symbolizes has occurred. In most languages, raising an exception also entails creating an exception object to hold the details for the exception. An example from Delphi:

```
raise Exception.Create('An error occurred');
```

As can be seen, the exception has a description of the error that occurred as a very prominent attribute. In most languages, programmers can define their own classes of exceptions, and associate attributes with them. In such a case, an exception might be raised like this:

```
raise EFileNotFound.Create('infile.dat', 'reading');
```

Although this is a bit of a contrived example, it shows that both the filename in question and the action desired on it can be encoded in an exception object. Usually, however, a descriptive message string suffices (since all desired extra attributes can simply be encoded in such a string).

At the moment an exception is raised, the normal execution of a program is aborted, and a search for a handler for the exception is started. If no handler is found, the entire program is aborted with an error condition. This kind of behaviour ensures that no program segments are executed if their preconditions are not satisfied (provided exceptions are thrown if this is the case).

Consider the following code fragment, where DoAction raises an exception if the call to it was not successful:

```
...

DoAction;
{ DoAction was successful }

...
```

[3] In C++, raising an exception is called *throwing* it. Conversely, handling a thrown exception is called *catching* it.

It is easy to see that the assertion *DoAction was successful* always holds: if DoAction is executed and succeeds, execution proceeds normally and the assertion is indeed true. However, should DoAction fail, it raises an exception. At that point, execution is aborted, and the location of the assertion is never reached, thus it is never evaluated. In fact, the assertion is only evaluated whenever it would evaluate to true.

This important fact tells us two more things about exceptions. First, success of a subprogram that uses exceptions for its error reporting can be easily determined from the fact that execution proceeds as usual, and failure can be determined from the fact that execution is aborted. Therefore, we do not need an extra return value to return the success result of a subprogram. This means we can go back to the classical notions of procedures and functions having distinct semantic meanings. Recall that we used to make every subprogram a function so it could easily return a success value – this "trick" is no longer necessary when exceptions are available.

Second, reconsider the example from section 2.2 where we had three subprograms, `operation1`, `operation2` and `operation3`, each depending on the successful prior execution of the previous one. Using traditional techniques we had to build elaborate branching structures around the operations to guard their execution. However, with exceptions guaranteeing that execution proceeds only if the subprograms are successful, we can simply concatenate the statements and have the exact same behaviour:

```
operation1(param1);
{ operation1 was successful }
operation2(param2);
{ operation2 was successful }
operation3(param3);
{ operation3 was successful }
{ all operations were successful }
```

Obviously, the comments are included only for clarity. Omitting them will make the code fragment a lot more concise and obviously elegant.

Assuming that the exceptions raised in each subprogram contain enough information for the user to determine the origin and cause of the erroneous condition, we don't even need to bother reflecting this information in calling programs, since the exceptions will provide that information at the moment they are handled. We will discuss handling exceptions in the next section.

3.3 Error handling through exceptions

However nice it is to be able to just concatenate statements and rest assured that each one will complete successfully before the next one is called, it is undesirable to have a program abort at the first error condition. That's why exceptions can be handled, by means of so-called *exception handlers*.

Most programming languages have two kinds of exception handlers. At least C++ and Delphi, the languages we discuss here, do. The two exception handlers reflect two different ways of responding to an exception being raised. Both handlers are block statements that guard a block of code, and respond in a specific way to exceptions that might occur inside the block. They are called **try-finally** and **try-except** handlers. Both handlers will be discussed here in their Delphi incarnation. C++ has equivalent handlers with slightly different but comparable syntax.

The **try-finally** handler is a block statement that looks like this:

```
try
     try_statements
```

```
finally
     finally_statements
end
```

Its semantics are as follows: first, the statements in `try_statements` are executed in the normal way. Upon exit of the block, the statements in `finally_statements` are *always* executed. What this means is that no matter how control leaves the try-block, by flowing off the end of the block or by an exception causing an abort, the statements in the finally-block are always executed. This exception handler is useful when allocated resources have to be guarded. Since in a language with exceptions there is no hard guarantee that any part of a statement sequence is ever reached, this might be a possible cause for memory leaks, as in the following example:

```
AllocateResources;
{ other statements }
ReleaseResources;
```

Since any of the intermediate statements contained in `{ other statements }` might raise an exception, causing abortion of the block, it is possible that the allocated resources are never freed. The solution is therefore to guard the resources with a **try-finally** block:

```
AllocateResources;
try
     { other statements }
finally
     ReleaseResources;
end;
```

This ensures that even if an exception is thrown in the try-block, the allocated resources are released before the exception is propagated to other handlers. This handler does not really provide any distinct improvement over traditional error handling code, since the same behaviour can be emulated using traditional control structures. However, in the face of exceptions and their potential to abort code, such a construct is required, and at the same time it does make the fact that a resource is being explicitly guarded clear to any later readers of the code. It does, therefore, increase code readability.

The other exception handler block, called a **try-except** block, follows the same form as the **try-finally** block:

```
try
     try_statements
except
     except_statements
end
```

Again, the statements in the try-block are guarded and the statements in the except-block are executed whenever an exception occurs. However, with this handler, the statements in the except-block are executed *only if* an exception occurs, and are assumed to handle the exception. After the execution of the except-block has finished (assuming no new exceptions are raised inside it), the exception that caused execution of the except-block is considered *handled* and will no longer be propagated. Execution will resume at the next statement after the *end*. This exception handler is designed to actually respond to error conditions, and it can be used in a variety of ways. It can be used to catch and ignore, or catch and correct error conditions. It can be used to show error information to the user, or to write such information to an error log. It can be used to guard resources when multiple resources need to be allocated atomically (a case in which **try-finally** would be inappropriate). When nested inside a loop, it can be used to retry subprogram calls until success is achieved. A full treatment of all possible patterns of error handling and resource

guarding falls outside the scope of this paper. I plan to write a later paper about this subject, however at the moment we will leave it at this.

Except-blocks do not need to handle all possible exceptions. Indeed, when rigorous exception handling is a goal, **try-except** blocks that handle everything are frowned upon. Generally, it is better to respond to known and handleable exceptions, and let unknown or unforeseen exceptions up to other handlers. Except-blocks can specify which exception types to catch, and what code to execute for each exception type. Uncaught exceptions are left unhandled and are propagated up the handler chain.

We have not yet discussed exception propagation. Whenever an exception is thrown, control passes to the closest enclosing exception handler. If this handler catches the particular type of exception, and successfully executes its except-block, the exception is considered handled and execution resumes at the next statement after the handler. If the exception is not handled, control passes to the next enclosing exception handler, and so forth, until the exception is either handled or the topmost exception handler has failed to handle the exception. At that point, control passes to a default exception handler of either the language or the operating system. At that point, both of these exception handlers have little other choice than aborting the program and displaying some indication of the error to the user.

The fact that exceptions are implicitly propagated from handler to handler with no regard for the code in between provides yet another advantage of exceptions over traditional error handling techniques: Intermediate subprograms that do not want to concern themselves with either error reporting or error handling do not need to contain any extra error-related code. This reduces code clutter and improves all qualitative factors of readability, writability and reliability of the code. With this in mind, we can rewrite the `initialize` routine from section 2.2 like this:

```
void initialize() {
     subsystem_init_step1();
     subsystem_init_step2();
}
```

The `initialize` routine itself need not be concerned which, if any, of the initialization calls fails. If one should fail, its raised exception will propagate through to any exception handler `initialize`'s caller will have prepared, where the failure to initialize will be handled accordingly.

3.4 Exception-based error handling summary

It is generally recognized that one of the greatest benefits of exceptions is reduced code clutter, and clear separation of error handling code from normal program code through except-blocks. Furthermore, all the problems with traditional error handling outlined in section 2 can be overcome by the use of exceptions.

To re-iterate, we will repeat the summary of problems from section 2.3 and present the solution that exceptions provide with each point.

Traditional error handling	Exception-based error handling
Distinction between procedures and functions is lost	This is not necessary, since errors are reported by ways other than return values.
Integer error values do not carry meaningful information	Exception objects carry all relevant information within them

Traditional error handling	Exception-based error handling
Nesting of several handlers detracts greatly from readability and modifiability	Handlers need only be nested when explicit handling is desired. Subprogram calls that depend on one another can simply be concatenated.
Error propagation must be explicitly implemented at every level of the subprogram call chain	Error propagation is implicit, and subprograms that do not handle or generate errors do not need to contain any error-related code
Traditional handling is unable to cope with certain classes of errors	Even exceptions generated outside the application can be handled in a uniform way inside it.

4 The benefits of traditional methods

Despite the apparent benefits of exception-based error handling, it still isn't for every case. Most notably, in the case of a "quick hack" program or script, error handling is usually left out altogether. In such a case, functions that employ traditional error-handling will behave reasonably in the face of error conditions, for example by returning or outputting zero, or an empty string, or performing no action. By contrast, a single exception would "crash" the program. Especially in a scripting environment, where the program is protected from hard crashes by things such as faulty pointers, it is an attractive prospect to simply fudge result codes. In case of error conditions, faulty output can easily be recognized and ignored by human users. What the user does not want, however, is the program bailing out.

It can be argued that leaving out error handling altogether is wrong, and in any case in such a quick hack program, inserting some extra code just to explicitly ignore exceptions is no big problem. In any case, it is still good to be knowledgeable of situations where exceptions might be more trouble than they're worth. It should be noted, however, that such situations are few and far between.

5 Conclusion

Ultimately, there is no single “right” construct for every job. Just as with almost every aspect of programming, or engineering in general, there is the concept of *the right tool for the right job*, meaning that no single solution is best in all cases[4]. It is therefore important to be aware of several alternatives, so that in the light of a problem to solve, the best applicable solution can be chosen.

We have seen that, in serious software projects of non-negligible size, where errors cannot just be “swept under the carpet”, exceptions can provide a significant benefit over traditional error handling techniques. They provide a simple, standardized, and elegant way of signaling and resolving error conditions. Usually, the same effect can be reached with less lines of code, and normal and error-resolving code is clearly separated. Furthermore, an application will be able to survive in the face of before irresolvable conditions.

The benefits of exceptions are steadily more recognized by the general audience. For example, a popular language for web scripting, PHP, started as a simple macro language. After it became popular, it started to evolve more in the direction of a full-fledged programming language, albeit still focused on development for the World Wide Web, and still very much aimed at quick-‘n-dirty development. Right now it is maturing up, and will even include support for exceptions in its next major release [4].

If I may speak from personal experience, learning to program with exceptions takes some practice, but once you get used to it, you will never want to do without them again.

6 References

[1] Robert W. Sebesta, *Concepts of programming languages*, 2004, Sixth Edition, International Edition, ISBN 0-321-20458-1, pages 8-18

[2] Example taken from the Win32 API. A reference of the method can be found here: http://msdn.microsoft.com/library/en-us/debug/base/getlasterror.asp

[3] Carlo Ghezzi, Mehdi Jazayeri, Dino Mandioli, *Fundamentals of Software Engineering,* 2003, Second Edition, International Edition, ISBN 0-13-099183-X, pages 17-20

[4] A modification list of the new PHP engine can be found here. Among them is a description of the new support for exceptions: http://www.php.net/zend-engine-2.php

[4] On a sidenote, if more people would realize this, we could put an end to the tiresome yet heated debates of which software system is better, of which the most well-known debates concern operating systems and programming languages.